# Ziria: Wireless Programming for Hardware Dummies

Gordon Stewart
Princeton

Mahanth Gowda
UIUC

Geoffrey Mainland
Drexel

Božidar Radunović
MSR

Dimitrios Vytiniotis
MSR

## Abstract

Software-defined radio (SDR) brings the flexibility of software to the domain of wireless protocol design, promising both an ideal platform for research and innovation and the rapid deployment of new protocols on existing hardware. However, existing SDR platforms either require careful hand-tuning of low-level code, negating many of the advantages of software, or are too slow to be useful in the real world.

In this paper we present Ziria, the first software-defined radio platform that is both easily programmable and performant. Ziria introduces a novel programming model tailored to wireless physical layer tasks and captures the inherent and important distinction between data and control paths in this domain. We describe an optimizing compiler for Ziria, provide a detailed evaluation, and give a line-rate Ziria implementation of 802.11a/g.

## 1. Introduction

The past few years have witnessed tremendous innovation in the design and implementation of wireless protocols, both in industry and academia (cf. [7, 21, 28]). Much of this innovation has occurred at the physical (PHY) layer of the protocol stack which manages the translation between radio hardware signals and protocol packets. These innovations have taken the form of numerous new signal processing algorithms and novel coding schemes, both of which have greatly increased the efficiency of existing radio communication channels.

Many of these innovations were first implemented using software-defined radio (SDR) platforms. Software-defined radio refers to wireless protocol design in which a protocol's complex signal processing is performed in software rather than in hardware. There are clear advantages to implementing novel protocol designs in software, such as ease of development, fast and cheap deployment, and a much shorter development cycle compared to a hardware implementation. For example, GnuRadio [2, 10], currently one of the most widely-used SDR platforms, is implemented in a combination of C++ and Python, meaning it is very easy to program and extend. Unfortunately, this extensibility and ease of use comes at a cost: GnuRadio suffers from serious performance limitations compared to hardware implementations (for example [25] reports bus transfer delays of hundreds of $\mu$s). Signal processing algorithms often require substantial processing power, and modern PHYs impose tight time constraints. For example, WiFi requires a receiver to process each signal sample in 25ns. Meeting these demands is challenging in general, and impossible with GnuRadio. These performance requirements make GnuRadio and similar platform of limited utility for testing real network deployments where line-speed operation is critical.

This is not to say that no SDR architecture provides acceptable performance: Warp [23], Sora [29], and TI KeyStone [1] are high-performance hardware-software SDR platforms. These platforms can meet tight timing deadlines, and thus provide "real-time" support for protocol designers wishing to test at line rate. However, they suffer from another problem, which has seriously limited their adoption: these platforms are difficult to program. FGPA-based platforms, such as WARP, require substantial digital design expertise. CPU and DSP-based platforms, such as Sora and TI KeyStone, require the ability to write code that is highly-tuned to the underlying processor's architecture. For example, Sora relies heavily on externally created lookup tables for performance. Furthermore, different parts of the receiver are manually placed onto different cores in order to balance CPU load. Such contortions are heavily hardware dependent and must be performed manually for each new architecture.

In this paper, we present a new language, Ziria, and corresponding programming model closes the gap between performance and flexibility. Ziria consists of two components: (1) a high-level domain-specific language for programming wireless protocols, and (2) an execution model and compiler transform Ziria programs to line-rate software radio implementations. In this way, Ziria combines the best features of extensible, highly programmable, but low performance platforms like GnuRadio, with those of high performance but difficult-to-program platform such as Sora. To demonstrate that our system supports designing real signal processing code, we present a Ziria implementation of WiFi 802.11a/g. The optimizations implemented in our compiler, which include automatic vectorization, automatic lookup table (LUT) generation, and annotation-guided pipeline parallelization, allow our Ziria implementation to generously meet the timing limits imposed by the 802.11a/g standards; its performance even approaches that of hand-optimized code. To the best of our knowledge, we are the first to present a *high-level* programming platform that can implement the 802.11a/g protocol on a general-purpose CPU while meeting timing constraints. In summary, our contributions are as follows:

- We present Ziria, the first software radio platform that provides a flexible, high-level programming model while meeting the timing constraints required for line-rate deployments.
- Our compiler for Ziria implements automatic vectorization, automatic lookup table generation, and annotation-guided pipelining, optimizations that are vital to performance and provide up to an order of magnitude speed-up over un-optimized code.
- To demonstrate the viability of Ziria as a platform for developing SDR applications, we implement 802.11a/g. Our implementation meets the timing constraints imposed by the protocol specification, and its performance approaches that of hand-optimized code.

In Ziria, all signal processing code aside from a few heavily used standard functions, e.g., Fast Fourier Transform (FFT) and Viterbi decoding, is written in the high-level language without reference to the underlying hardware architecture. This leads to a programming model in which Ziria programs closely resemble the original protocol specifications. Nonetheless, the programmer does not have to sacrifice line-rate speed to gain this clarity, as our benchmarks show. Indeed, it is the very high-level nature of Ziria that gives our optimizer the opportunity to transform code into an efficient, low-level implementation.

## 2. Ziria by example

In this section we give a high-level overview of the main components of the Ziria language and its execution model. We then examine fragments of our 802.11 implementation: first, the imperative innards of a Ziria scrambler block, which is used in our WiFi transmitter to XOR input packet data with a pseudorandom sequence in order to shape the transmitted signal, and then the outer pipeline of our WiFi 802.11a/g receiver. We introduce necessary signal processing concepts along the way.

### 2.1 Language and execution model

Execution in Ziria is centered around *stream processing*: reading values from a (potentially infinite) input stream and writing values to an output stream. For example, at a high level, a WiFi receiver is just a computation that reads a stream of complex numbers from the A/D unit of a radio and outputs a stream of MAC-layer packets.

***Language*** Ziria provides both traditional stream processors, which map values from input to output streams and which never terminate, which we call *stream transformers*, and *stream computers*, a novel stream processing language construct. Figure 1 depicts both stream transformers and stream computers. On the left, two transformers are composed vertically into a two-stage pipeline. On the right, a stream computer is composed with a stream transformer.

Like stream transformers, stream computers map stream inputs to stream outputs. However, unlike stream transformers, computers need not execute indefinitely. Instead, they execute for a while, consuming input and producing output, and then halt and return a *control value*. In Figure 1, this is depicted as the fat "Control" arrow connecting the computer to the transformer. If a stream computer is at the outermost position in a Ziria program, then its return value is just the program's return value. Otherwise, the control value is used to dynamically reconfigure the rest of the processing pipeline; in the figure, this corresponds to switching from the "Computer" to the "Transformer" in the lower right corner. After reconfiguration, the inputs that originally flowed to the computer are routed to the next component in the pipeline—in this case, the "Transformer." Both components output to the same stream. This dynamic reconfiguration directly reflects the control flow of many PHY-layer protocols, which read a preamble or packet header from the input stream and then reconfigure computation of the rest of the stream based on data in the preamble.

As an example, consider two Ziria stream computer primitives, emit and take. emit takes an expression $e$ and immediately writes its value to the output stream while returning the unit control value, $()$. take is computer that pulls a single value from the input stream and immediately halts computation, returning the input as a control value. These basic building blocks are composed in Ziria to perform arbitrary stream processing using Ziria's *bind* operator, which has the notation $x \leftarrow c_1;\ c_2$. Bind runs the stream computation $c_1$ until it produces a return value $v$, and it then runs the dynamically configured stream processor $c_2[v/x]$ ($c_2$ with $v$ substituted for $x$). For example, the following is a small Ziria program that maps a function f over an input stream of x's, emitting a stream of results, f(x), to the output stream.

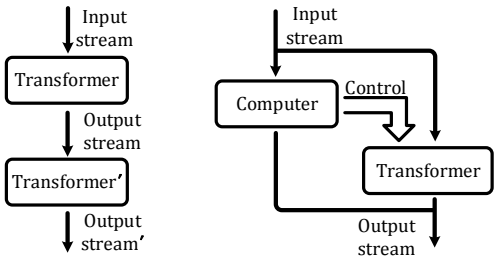



**Figure 1.** Transformer-transformer composition (left); computer-transformer composition (right).

```
repeat (x ← take; emit(f(x)))
```

The semicolon between take and emit is bind. It produces a program that behaves like take for one step, until take returns a control value x, then behaves like emit for one step, until emit yields the value f(x) onto the output stream. In our small example, this process is repeated indefinitely using Ziria's repeat combinator, which converts a stream computer (x←**take**; **emit**(f(x))) into a stream transformer by reinitializing the computer every time it returns a control value. Note that x←**take** binds the control result of take, namely x, in **emit**(f(x)). Bind is similar to the switch operator in Yampa [13]; however Yampa conflates control and data paths, whereas Ziria carefully maintains a distinction between the two.

Dataflow composition in Ziria is achieved by sequencing computations with the arrow combinator, $\ggg$. This form of composition is similar to standard stream transformer composition operators in dataflow languages such as StreamIt [31]. It takes a Ziria computation that maps stream inputs to stream outputs and routes its output to the input stream of a second Ziria computation. If either computation returns a control value, then the entire compound computation returns that control value. As illustration, consider the following Ziria program, which computes the square root of -1 and then immediately returns.

```
emit(-1) ⋙ repeat (x ← take; emit(sqrt(x)))
```

This program yields the value -1 onto an intermediate stream (**emit**(-1)), pulls the -1 off the intermediate stream (x ←**take**), and then is dynamically reconfigured to **emit**(sqrt(-1)), which prints sqrt(-1) to the output stream and returns unit.

***Execution model*** At runtime, each Ziria computation is implemented as a pair of functions, called "tick" and "process."[1] We consider process first. A computation's "process" function takes a single argument, a value from the Ziria program's input stream, performs the required computation on the input value, and returns a *result* of an enumerated type in $r$. Results $r ::=$ skip | yield($v$) | done($v$) are either yield($v$), indicating that the value $v$ is ready to be written to the program's output stream, skip, indicating that the program should be called again but that no output is immediately available, or done($v$), indicating successful program termination with a returned control value $v$. Note that only stream computers may produce a result done($v$); transformers must either skip or yield.

Tick is used to drive computation of blocks that do not require values from their input stream in order to do useful work. For example, the Ziria program **emit** 1; **emit** 2 writes the sequence

[1] Because our compiler generates code in continuation-passing style, "tick" and "process" are actually code labels, and calls to these functions are actually jumps. These details are not important here; see Section 4 for a more in-depth discussion.

1,2 onto its output stream but does not require any input to do so. The tick function returns an enumerated type called a *result kind*. Result kinds $k ::= \texttt{imm}(r) \mid \texttt{cons}$ are either imm, for "immediate," indicating that the current computation is ready either to yield an output value or to return a done value, or cons, which indicates that the computation needs to consume input to proceed. The outer loop of every Ziria program first ticks the computation, processing immediate results along the way, until either cons or $\texttt{imm}(\texttt{done}(v))$ is returned. On cons, the outer loop proceeds to call the program's process function with a value from the input stream, handles the result, then returns to calling tick once again.

In compound blocks such as $x \leftarrow c_1;\ c_2$ and $c_1 \ggg c_2$, the computation $c_1$ to the left of the bind or arrow combinator need not return an immediate value directly to the driver loop. In $c_1 \ggg c_2$, $c_1$ yields stream output values by immediately calling the process function of $c_2$. We use a similar strategy when compiling $x \leftarrow c_1; c_2$: control values produced by $c_1$ are written to a statically allocated private buffer shared by $c_1$ and $c_2$, and references to the variable $x$ in $c_2$ are translated to reads from this shared buffer.

### 2.2 WiFi 802.11a/g: TX scrambler and RX pipeline

In order to illustrate the versatility of Ziria, this section walks through two examples of real Ziria code. The first, a signal scrambler, is typical of the kind of imperative code that lives within the blocks of a Ziria pipeline. The second example is the full pipeline of our 802.11a/g receiver.

***Scrambler*** In signal processing domains, the purpose of a scrambler is to XOR input data with a pseudorandom sequence. This reduces the probability of sending data sequences that have undesirable signal properties, such as all 1's or all 0's, over the air (cf. Section 17.3.5.4 of [19]).

```
1  let scrambler() =
2    letref scrmbl_st: arr[7] bit := {1,1,1,1,1,1,1};
3           tmp: bit; y: bit;
4    in repeat (
5      x ← take;
6      return (
7        tmp := (scrmbl_st[3] ^ scrmbl_st[0]);
8        scrmbl_st[0:5] := scrmbl_st[1:6];
9        scrmbl_st[6] := tmp;
10       y := x ^ tmp);
11     emit (y))
```

**Listing 1.** Scrambler function of WiFi 802.11a/g transmitter in Ziria

The scrambler above, a let-bound Ziria computation taking no arguments, is an example of a feedback shift register. The scrambler's body declares three local variables in lines 2 through 3: scrmbl_st, an array of 7 bits that gives the current state of the shift register, and two one-bit references: tmp and y. The scrambler **take**s a value from the input stream (line 5), then performs an imperative computation that assigns tmp the XOR of taps 3 and 0 in the shift register, shifts the register state left by one, feeds tmp into position 6 of the register, and finally returns the XOR of tmp and the input bit x.

There are two interesting aspects to this code. First, because both the standard and the code in the listing operate on bit arrays, one can easily verify by inspection that the listing code matches the definitions found in the WiFi standard. This would be more difficult if we implemented the scrambler directly in a more efficient fashion, say by using an integer rather than a bitvector to store the scrambler state, and by shifting instead of indexing into the array. Second, we remark that, when compiled with the Ziria compiler, the scrambler in Listing 1 compiles to quite efficient code. In the context of our WiFi pipeline, the Ziria compiler first automatically generates a lookup table for the scrambler (cf. Section 4.3), then vectorizes the code to operate on multiple inputs at a time (cf. Section 4.2).

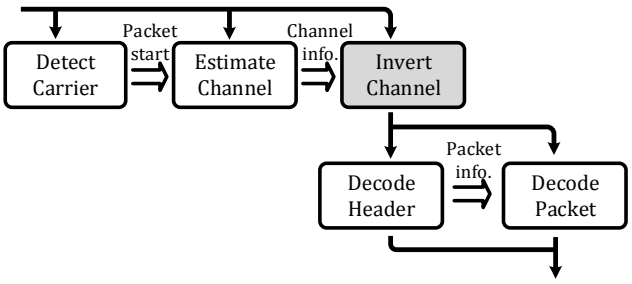



**Figure 2.** Schematic representation of a WiFi receiver. The shaded box is a transformer; the white boxes are computers.

This gives a $14.8\times$ speedup (cf. Section 5) over the same code compiled without optimizations. As comparison, in the current Sora implementation, the same scrambler is defined as a hand-written lookup table, an excerpt of which is shown in Listing 1.

```
1  const unsigned char
2  SCRAMBLE_11A_LUT[SCRAMBLE_11A_LUT_SIZE] = {
3      0x00, 0x91, 0x22, 0xb3, 0x44, 0xd5, 0x66, 0xf7,
4      0x19, 0x88, 0x3b, 0xaa, 0x5d, 0xcc, 0x7f, 0xee,
5      //... 13 lines elided ...
6      0x87, 0x16, 0xa5, 0x34, 0xc3, 0x52, 0xe1, 0x70,};
```

**Listing 2.** Sora lookup table corresponding to the Ziria scrambler implementation in Listing 1

The lookup table implementation may be efficient, but it gives the reader no insight into the computation being performed. Nor is it easy to verify by inspection that this implementation meets the scrambler specification given in the WiFi standard.

***WiFi receiver*** The next code example illustrates the other end of the Ziria spectrum: a complete signal processing pipeline for an 802.11a/g receiver. This pipeline consists of a number of signal processing components, a block diagram for which is given in Figure 2. We first describe what each component does, then take a closer look at the corresponding code, which is given in Listing 3.

The first block, "Detect Carrier," determines whether a WiFi transmitter is operating on the radio channel by looking for a known constant preamble sequence. Once carrier detection observes the preamble of a valid packet transmission, the pipeline enters a channel estimation phase, operating over the subsequent 160 input samples, in which it attempts to quantify physical effects such as multipath fading on the transmitted signal. The channel information is then used in the channel inversion block of the steady state of the pipeline (in gray) to nullify these effects. This block feeds input data first to a block that decodes the packet header, then to a block that decodes the packet payload. Information from the header decoding phase such as the data rate is used to configure the packet decoding stage.

```
1  let detectCarrier() = CCA() in
2  let channelEstimation = t11aLTS() in
3  let invertChannel(cInfo:int) =
4    t11aDataSymbol() ⋙ t11aFreqCompensation() ⋙
5    t11aFFT() ⋙ t11aChannelEqualization(cInfo) ⋙
6    tPhaseCompensate() ⋙ tPilotTrack()
7  in read ⋙ downSample() ⋙ (
8     detectCarrier();
9     cInfo ← channelEstimation();
10    invertChannel(cInfo) ⋙ (
11       pInfo ← t11aDecodePLCP();
12       t11aDecode(pInfo))) ⋙ write
```

**Listing 3.** Top-level Ziria pipeline for WiFi 802.11a/g receiver

The Ziria code corresponding to the pipeline of Figure 2 is given in Listing 3. The structure of this code follows that of the block diagram quite closely. After reading from the input stream and downsampling (line 7), the first operation performed is carrier detection (line 7). Here, `detectCarrier()` is an alias introduced with a let-binding (line 1) for another Ziria function, `CCA()` or clear channel assessment. Carrier detection is sequenced using the bind operator with channel estimation, a Ziria stream computer that returns a channel information value `cInfo`. The channel information `cInfo` is passed as an argument to `invertChannel`, a Ziria function defined at line 3. Finally, we stream the output of channel inversion to a Ziria computer that first decodes the packet header (`t11aDecodePLCP()`), and then decodes the packet payload (`t11aDecode(pInfo)`).

## 3. Language

In this section, we present more details about the Ziria language and the key ideas behind its type system and operational semantics.

### 3.1 Syntax

The key abstraction in Ziria is that of stream computations $c$, whose syntax is given in Figure 3. We already presented in Section 2 the primitive combinators take, emit, and repeat. The return combinator trivially evaluates its argument and returns it on the control channel. The $\mathsf{map}(f)$ combinator applies $f$ to every value in the input stream, emitting the result onto the output stream.

As explained in Section 2, primitive combinators can be composed along the control or the data path with Ziria's composition operators bind ($x \leftarrow c_1;\ c_2$) and arrow ($c_1 \ggg c_2$), respectively. In addition to these constructs, Ziria includes local functions that can bind computations (letfunc) and computation function applications $c(\overline{e})$. It also allows for mutable computation-local state, with the letref construct. We will return to the use of shared state between processing pipeline components throughout the rest of the paper, since the careful treatment of state in Ziria is key to a rich set of optimizations that dramatically improve performance. The conditional computation if $e$ then $c_1$ else $c_2$ configures the computation to be either $c_1$ or $c_2$ depending on the value of $e$. More conventionally, Ziria incorporates a sub-language of imperative expressions ($e$) and functions (letfun) that can be used in computations such as $\mathsf{map}(f)$ and emit $e$. This imperative sub-language includes constructs for manipulating primitive types such as bits, complex numbers, and arrays. The details are not particularly interesting.

The Ziria type language is stratified into stream (ST) types $\sigma$ and expression types $\tau$. Stream types describe stream computations. For example, a stream transformer computation with type $\mathsf{ST}\ \mathsf{T}\ a\ b$ (e.g. $\mathsf{map}(f)$) executes indefinitely, transforming a stream of values of type $a$ to a stream of $b$'s. As we originally explained in Section 2, stream computers of type $\mathsf{ST}\ (\mathsf{C}\ \nu)\ a\ b$ are similar, except that at some point during stream processing they may conclude the computation by returning a value of type $\nu$. The language of expression types is standard, and includes a rich collection of base types, arrays, etc. We remark that the type language disallows arbitrary higher-order functions and partial applications in well-typed Ziria programs, to avoid closure allocation costs. Finally streams may only contain values with $\tau$ types.

### 3.2 Type system

The Ziria type system (Figure 4) includes a judgment form for typing computations $\Gamma; \Delta; \Sigma \vdash c : \sigma$ as well as a judgment (not shown) for typing imperative expressions $\Gamma; \Sigma \vdash e : \tau$. The typing rules make use of three type variable contexts. $\Gamma$ maps variables to expression types $\theta$, $\Delta$ maps computation variables to computation types $\xi$, and the store typing $\Sigma$ maps mutable variables to base

***Stream computations and imperative expressions***

| | | |
|---|---|---|
| $c ::=$ | $x \leftarrow c;\ c$ | Bind |
| | $c \ggg c$ | Arrow |
| | letref $\overline{x{:}\tau := v}$ in $c$ | Scoped shared state |
| | letfun $g(\overline{x{:}\tau}) = m$ in $c$ | Local function |
| | letfunc $f(\overline{x{:}\tau}) = c_1$ in $c_2$ | Local computation |
| | $c(\overline{e})$ | Call computation function |
| | take | Move to control channel |
| | emit $e$ | Emit on data channel |
| | return $m$ | Return on control channel |
| | repeat $c$ | Repeat $c$ indefinitely |
| | $\mathsf{map}(f)$ | Map over input stream |
| | if $e$ then $c_1$ else $c_2$ | Conditionals |
| | $\ldots$ | |
| $e, m ::=$ | $x \mid v \mid x := e \mid m_1; m_2 \mid f(\overline{e}) \mid \ldots$ | Expressions |
| $v ::=$ | $() \mid i \mid \ldots$ | Values |

***Computation and expression types***

| | | |
|---|---|---|
| $\xi ::=$ | $\sigma \mid \overline{\tau} \to \sigma$ | Computation types |
| $\sigma ::=$ | $\mathsf{ST}\ (\mathsf{C}\ \tau)\ \tau\ \tau \mid \mathsf{ST}\ \mathsf{T}\ \tau\ \tau$ | Computers/Transformers |
| $\theta ::=$ | $\tau \mid \overline{\tau} \to \tau$ | Expression types |
| $a, b, \tau, \nu ::=$ | unit \| int \| complex \| $\ldots$ | Base types |

**Figure 3.** Syntax of Ziria

$$\boxed{\Gamma; \Delta; \Sigma \vdash c : \sigma}$$

$$\frac{\Gamma; \Delta; \Sigma \vdash c_1 : \mathsf{ST}\ (\mathsf{C}\ \nu)\ a\ b \quad \Gamma, x : \nu; \Delta; \Sigma \vdash c_2 : \mathsf{ST}\ t\ a\ b}{\Gamma; \Delta; \Sigma \vdash x \leftarrow c_1; c_2 : \mathsf{ST}\ t\ a\ b} \qquad \frac{\Gamma; \Delta; \Sigma \vdash c : \mathsf{ST}\ (\mathsf{C}\ \mathsf{unit})\ a\ b}{\Gamma; \Delta; \Sigma \vdash \mathsf{repeat}\ c : \mathsf{ST}\ \mathsf{T}\ a\ b}$$

$$\Gamma; \Delta; \Sigma \vdash \begin{cases} \mathsf{take} : & \mathsf{ST}\ (\mathsf{C}\ a)\ a\ b & \\ \mathsf{emit}\ e : & \mathsf{ST}\ (\mathsf{C}\ \mathsf{unit})\ a\ \tau, & \text{if } \Gamma; \Sigma \vdash e : \tau \\ \mathsf{return}\ m : & \mathsf{ST}\ (\mathsf{C}\ \tau)\ a\ b, & \text{if } \Gamma; \Sigma \vdash m : \tau \\ \mathsf{map}(f) : & \mathsf{ST}\ \mathsf{T}\ a\ b, & \text{if } (f{:}a \to b) \in \Gamma \end{cases}$$

$$\frac{\Sigma \propto (\Gamma_1, \Sigma_1) \oplus (\Gamma_2, \Sigma_2) \quad \Gamma\Gamma_1; \Delta; \Sigma_1 \vdash c_1 : \mathsf{ST}\ t\ a\ b \quad \Gamma\Gamma_2; \Delta; \Sigma_2 \vdash c_2 : \mathsf{ST}\ \mathsf{T}\ b\ c}{\Gamma; \Delta; \Sigma \vdash c_1 \ggg c_2 : \mathsf{ST}\ t\ a\ c} \qquad \frac{\Sigma \propto (\Gamma_1, \Sigma_1) \oplus (\Gamma_2, \Sigma_2) \quad \Gamma\Gamma_1; \Delta; \Sigma_1 \vdash c_1 : \mathsf{ST}\ \mathsf{T}\ a\ b \quad \Gamma\Gamma_2; \Delta; \Sigma_2 \vdash c_2 : \mathsf{ST}\ t\ b\ c}{\Gamma; \Delta; \Sigma \vdash c_1 \ggg c_2 : \mathsf{ST}\ t\ a\ c}$$

**Figure 4.** Computation typing ($t \in \{(\mathsf{C}\ \nu), \mathsf{T}\}$)

expression types $\tau$. Type checking the access to a mutable variable looks up the variable in the store typing $\Sigma$, whereas accessing a function argument looks it up in $\Gamma$ (for read-only variable).[2]

We discuss some interesting typing rules of Ziria, given in Figure 4. The typing judgment for bind ($x \leftarrow c_1;\ c_2$) requires that $c_1$ is a stream computer taking streams of $a$'s to streams of $b$'s, potentially returning a value of type $\nu$. The $c_2$ component has a stream type $\mathsf{ST}\ t\ a\ b$ in the extended context $\Gamma, x{:}\nu; \Delta$. Note that $c_2$ may be *either* a stream computer or transformer, but its type determines the type of bind. The rule for repeat $c$ types it as a stream transformer when $c$ is a computer that returns $()$. take has type $\mathsf{ST}\ (\mathsf{C}\ a)\ a\ b$ since it takes a value of type $a$ from the input stream and immediately returns it. It is polymorphic in the output queue and thus can be composed using $\ggg$ with stream computations of arbitrary input type $b$. emit $e$ is a stream computer that evaluates an expression $e : \tau$ and yields it onto the output stream, returning the unit value $()$ ($\mathsf{ST}\ (\mathsf{C}\ \mathsf{unit})\ a\ \tau$). emit can be composed using $\ggg$ to the right of

[2] The full language also includes a let-binding in the computation and expression languages for introducing immutable local variables. The types of these variables are tracked in $\Gamma$.

stream computations with arbitrary output types $a$. Finally, return lifts an expression $m : \tau$ into the language of stream computations ($\mathsf{ST}\ (\mathsf{C}\ \tau)\ a\ b$) by evaluating it and returning it on the control path.

The typing rules for the $\ggg$ combinator are slightly more involved. The arrow combinator ($\ggg$) can appear overloaded, and can be typed with two typing rules. The first says that $c_1 \ggg c_2$ has type $\mathsf{ST}\ t\ a\ c$ if $c_1$ is a stream computation (transformer or computer) taking $a$'s to $b$'s ($\mathsf{ST}\ t\ a\ b$) and $c_2$ is a stream *transformer* ($\mathsf{ST}\ \mathsf{T}\ b\ c$) taking $b$'s to $c$'s. The second $\ggg$ rule is symmetric.

The $\ggg$ rules type components $c_1$ and $c_2$ of $c_1 \ggg c_2$ in different contexts in order to prevent communication through shared state. Imagine that $c_1$ and $c_2$ communicate via a buffered channel. An element that has been produced by $c_1$ under a particular view of the shared state and emitted on the communication channel may eventually get processed by $c_2$ under a different view of the shared state, since $c_1$ may have updated the state in the meantime. Under such conditions the semantics of $c_1 \ggg c_2$ becomes difficult to reason about—in fact, we will see in Sections 4.2 and 4.4 that the presence of shared state invalidates two important optimizations. To avoid these problems we split the store typing $\Sigma$ into two pairs $(\Gamma_1, \Sigma_1)$ and $(\Gamma_2, \Sigma_2)$, to ensure that no computation writes to state that the other component either reads or writes. For reasons of space we give the rules of this relation just for single-variable contexts:

$$\frac{}{(x{:}\tau) \propto ((x{:}\tau), \cdot) \oplus ((x{:}\tau), \cdot)} \qquad \frac{}{(x{:}\tau) \propto (\cdot, (x{:}\tau)) \oplus (\cdot, \cdot)}$$

$$\frac{}{(x{:}\tau) \propto (\cdot, \cdot) \oplus (\cdot, (x{:}\tau))}$$

### 3.3 Reference semantics

Ziria programs compile to an IR in C. We now give a semantics that captures the essence of the IR execution model. In addition to the tick/process-based execution (outlined in Section 2), each component is equipped with an *initialization* function.

A Ziria computation becomes *activated* through initialization (judgment $\Rightarrow$, Figure 6), meaning that it is ready to participate in the main tick/process loop. The judgment $S; c \Rightarrow S'; h$ initializes the computation $c$ in state $S$. Arbitrary code may be executed during initialization, hence the state $S$ may be updated to $S'$. A bind $(x \leftarrow h_1;\ c)$ that is activated contains an activated first component $h_1$, ready to be ticked. $h_1 \ggg h_2$ is the activated version of $c_1 \ggg c_2$ in which both components are activated. The activated computation $\{\!|L, h|\!\}$ arises from local letref definitions. Simple primitives are activated immediately but note that emit $v$ and return $v$ only mention values. The initialization of components that include expressions forces their evaluation, and this in turn may cause state updates. Finally, repeat $c$ initializes by activating $c$ to $h$ and reconfiguring to $(h; \mathsf{repeat}\ c)$, which allows the runtime to subsequently tick the first component $h$.

***Runtime*** Once a computation $c$ has been initialized, it can participate in the main loop. We model the runtime execution in Figure 5. The $\hookrightarrow$ judgment steps triples of an input buffer $I$, an activated computation $h$, and an output component $O$ to a new configuration. The intuition behind the rules in Figure 5 is that we first try to tick a component (using $\Uparrow$) and if that returns a result kind of `cons`, we peel a value off the input stream and push it through the computation's process function using $\Downarrow$. Results are emitted onto the output stream. To make the presentation shorter, the $\hookrightarrow$ relation has no case for $\mathtt{done}(v)$ values—we assume that the top-level computation we evaluate is a stream transformer and hence does not return.

***Tick and process*** As described in Section 2, the process function of a Ziria computation returns a result $r$, either skip, yield $v$, or done, while tick returns a result kind $k$: either an immediate result (imm $r$) or a request for more input (consume). Figure 5 gives the precise definition of $\Uparrow$ (tick) and $\Downarrow$ (process) for some of the more interesting Ziria computations.

***Runtime semantics*** $\boxed{I; S; h; O \hookrightarrow I'; S'; h'; O'}$

$$\frac{S; h \Uparrow S'; h'; \mathtt{imm}(\mathtt{skip})}{I; S; h; O \hookrightarrow I; S'; h'; O} \qquad \frac{S; h \Uparrow S'; h'; \mathtt{imm}(\mathtt{yield}(v))}{I; S; h; O \hookrightarrow I; S'; h'; v{:}O}$$

$$\frac{S; h \Uparrow S'; h'; \mathtt{cons} \quad S'; h'; v \Downarrow S''; h''; \mathtt{skip}}{(v{:}I); S; h; O \hookrightarrow I; S''; h''; O} \qquad \frac{S; h \Uparrow S'; h'; \mathtt{cons} \quad S'; h'; v \Downarrow S''; h''; \mathtt{yield}(u)}{(v{:}I); S; h; O \hookrightarrow I; S''; h''; (u{:}O)}$$

**Figure 5.** Runtime semantics

***Basic definitions***

$$\begin{array}{rcll} r & ::= & \mathtt{skip} \mid \mathtt{yield}(v) \mid \mathtt{done}(v) & \text{Results} \\ k & ::= & \mathtt{imm}(r) \mid \mathtt{cons} & \text{Result kinds} \\ S, L & ::= & \cdot \mid S, x \mapsto v & \text{Stores} \\ h & ::= & \{\!|L, h|\!\} \mid (x \leftarrow h_1;\ c_2) & \text{Activated computations} \\ & \mid & h_1 \ggg h_2 \mid \mathsf{take} \mid \mathsf{emit}\ v \mid \mathsf{return}\ v \mid \mathsf{map}(f) & \end{array}$$

***Computation initialization (excerpt)*** $\boxed{S; c \Rightarrow S'; h}$

$$\frac{S; c_1 \Rightarrow S'; h_1}{S; (x{\leftarrow}c_1;\ c_2) \Rightarrow S'; (x{\leftarrow}h_1;\ c_2)} \qquad \frac{S; c_1 \Rightarrow S'; h_1 \quad S'; c_2 \Rightarrow S''; h_2}{S; (c_1 \ggg c_2) \Rightarrow S''; (h_1 \ggg h_2)}$$

$$\frac{S; c \Rightarrow S'; h}{S; \mathsf{repeat}\ c \Rightarrow S'; (h; \mathsf{repeat}\ c)} \qquad \frac{S, \overline{x \mapsto v}; c \Rightarrow S', h}{S; \mathsf{letref}\ \overline{x{:}\tau := v}\ \mathsf{in}\ c \Rightarrow S'; \{\!|\overline{x \mapsto w}, h|\!\}}$$

**Figure 6.** Computation initialization, used in Figure 7

***Tick (excerpt)*** $\boxed{S; h \Uparrow S'; h'; k}$

$$\frac{S; h_1 \Uparrow S'; \_; \mathtt{imm}(\mathtt{done}(v)) \quad S'; c_2[v/x] \Rightarrow S''; h_2 \quad S''; h_2 \Uparrow S'''; h_2'; k}{S; (x \leftarrow h_1;\ c_2) \Uparrow S'''; h_2'; k}$$

$$\frac{S; h_2 \Uparrow S'; h_2'; \mathtt{cons} \quad S'; h_1 \Uparrow S''; h_1'; \mathtt{imm}(\mathtt{yield}(v)) \quad S''; h_2'; v \Downarrow S'''; h_2''; r}{S; h_1 \ggg h_2 \Uparrow S'''; h_1' \ggg h_2''; \mathtt{imm}(r)}$$

$$\frac{}{S; \mathsf{emit}\ v \Uparrow S; \mathsf{return}\ (); \mathtt{imm}(\mathtt{yield}(v))} \qquad \frac{}{S; \mathsf{return}\ v \Uparrow S; L; \mathsf{return}\ v; \mathtt{imm}(\mathtt{done}(v))}$$

***Process (excerpt)*** $\boxed{S; h; v \Downarrow S'; h'; r}$

$$\frac{S, L; h; v \Downarrow S', L'; h; r}{S; \{\!|L, h|\!\}; v \Downarrow S'; \{\!|L', h'|\!\}; r} \qquad \frac{S; h_1; v \Downarrow S'; \_; \mathtt{done}(w) \quad S'; c_2[w/x] \Rightarrow S''; h_2}{S; (x \leftarrow h_1;\ c_2); v \Downarrow S''; h_2; \mathtt{skip}}$$

$$\frac{S; h_1; v \Downarrow S'; h_1'; \mathtt{yield}(w) \quad S'; h_2; w \Downarrow S''; h_2'; r}{S; h_1 \ggg h_2; v \Downarrow S''; h_1' \ggg h_2'; r} \qquad \frac{}{S; \mathsf{take}; v \Downarrow S; \mathsf{return}\ v; \mathtt{done}(v)}$$

**Figure 7.** Tick and process semantics

The $\Uparrow$ judgment takes an initialized $h$ and a store $S$ and maps these to a new computation $h'$, a new store $S'$, and a result kind $k$. The $\Downarrow$ judgment takes an $h$ that has requested data to process, an input value $v$, and an initial store $S$, and returns a new $h'$, an updated store $S'$, and a result $r$. The topmost tick rule for bind defines what happens when the computation $h_1$ returns with imm (done $v$) (with possible side effects to $S$): First, we reconfigure $c_2$ by substituting

```
(†) repeat (x←take;emit(e)) ⟹ letfun f(x) = e in map(f)
(∗) x←return e; c           ⟹ let x = e in c
(∗) return e; c             ⟹ let _ = e in c
times e1 i (return e2)      ⟹ return (for i in (0,e1) e2)

    (x←letfun f(..) = e in c1); c2
⟹  letfun f(..) = e in (x←c1; c2)

    letfunc f(..) = return e in ... f(..) ...
⟹  letfun f(..) = e in ... return (f(..)) ...

    (if e then c1 else c2) >>>c3
⟹  if e then c1>>>c3 else c2>>>c3
```

**Figure 8.** Selected rewrite steps performed by the Ziria optimizer

$v$ for $x$ ($c_2[v/x]$) and initialize it to $h_2$. Then we tick $h_2$. Note how we have discarded the local state of $h_1$ as it is no longer needed.

Next consider the tick rule shown for arrow ($\ggg$). While we only show one of four rules for $\ggg$ here due to space constraints, all four $\ggg$ rules together encode a single pattern of computation: When we tick the computation $h_1 \ggg h_2$, we always start by ticking $h_2$ first. This is necessary if $h_2$ is a computation like emit that yields values to the output stream without requiring any input. For example, $c_1 \ggg \mathsf{repeat}\ (\mathsf{emit}(3))$ is a computation that after initialization will tick indefinitely, producing a stream of 3's and ignoring $c_1$. Ticking from right-to-left means that we need no intermediate buffering between sub-computations, since we always drain the pipeline before requesting additional input. The other three tick rules for $\ggg$ which we have not shown here encode the situations in which (1) $h_2$ requires input and $h_1$ has an input to push; (2) $h_2$ requires input and $h_1$ ticks to an immediate result that is skip or done but not yield, in which case the whole computation returns that result; or (3) both $h_1$ and $h_2$ require input, in which case the entire computation returns consume. The process rules for $\ggg$ are similar except that instead of ticking from right to left, we process a value through the pipeline from left to right (*i.e.*, first $h_1$ then $h_2$).

The tick and process rules for base combinators are mostly straightforward. Ticking emit $e$ immediately yields $e$'s value $v$ and steps to return $()$, returning the unit value the next time the computation is ticked. Ticking take immediately returns consume while processing take with input $v$ just returns done $v$. There are no process rules for emit and return because the runtime semantics will never call process on these components. The process rule for take converts the take to a return and produces done $v$ as result.

# 4. Compiler

Compiling Ziria, as opposed to implementing it as a library, means that we can apply optimizations to the actual Ziria code, for better performance. This section details these optimizations, which include automatic vectorization, automatic lookup table (LUT) generation, and annotation-guided pipelining. Section 5 measures performance of each optimization via microbenchmarks and for WiFi pipelines.

The Ziria compiler has a standalone frontend, including a parser and typechecker. The frontend generates code in an explicitly typed intermediate language. The code generator outputs C code in continuation-passing style, which allows us to replace calls to the tick, process, and initialization functions with direct jumps to labels.

## 4.1 High-level optimizations

The Ziria compiler implements a series of type- and semantics-preserving optimizations that eliminate or fuse computations together in order to decrease the amount of processing that occurs across computations. This means less copying and fewer jumps to the outer runtime loop. Figure 8 gives some of the rewrite rules that the Ziria compiler implements. Particularly interesting are the rewrite rules (marked $*$) that convert **return** statements to **let** definitions and the *auto-map* transformation (marked †), which replaces a repeated **take**-**emit** block with a single **map** transformer. For example, in our scrambler implementation from Listing 1, the auto-map transformation allows us to extract the scrambler imperative code (lines 6 through 10) into a let-bound function definition `f` and replace the entire body of the scrambler component with a call to **map**(`f`). This transformation is important for producing performant code, as the overhead measurements in Section 5.2 show.

Other optimizations justified by the runtime semantics of Section 3 include the conversion of computation loops [3] to for-loops, inlining and loop unrolling, and floating definitions and conditionals out of computations to enable other optimizations like auto-mapping. Note that the correctness of pushing conditionals above $\ggg$ relies on the state invariants that we decribed in Section 3.

## 4.2 Vectorization

Wireless protocols are designed and specified so that each basic component operates at an intuitive data granularity. For example, in Section 2 we saw that in each step of computation, the scrambler of Listing 1 takes a single bit from its input queue and emits a single bit to its output queue. This implementation matches the granularity of the specification quite closely. Unfortunately, the tight timing deadlines of most wireless protocols mean we do not always have the luxury of sacrifing performance by, *e.g.*, packing one bit per char or operating on only one 32-bit precision complex number at a time on a 64-bit CPU. Our pipelines would be much more efficient if we were able to process arrays of elements simultaneously. Ideally, we want to *batch* the inputs and the outputs of components.

However, this is not straightforward. Different components operate at different data granularities. For example, the Wifi scrambler is followed by an encoder. Depending on the selected data rate, the encoder will be configured to one of three variants which take 1, 2, or 3 input bits, and produce 2, 3, or 4 output bits, respectively. In order to correctly vectorize, we must ensure that the vectorization of the scrambler is matched to the vectorization of all possible variants of the encoder. Otherwise, a component may terminate with unemitted data. Matching granularities in a large program is tedious and has been done manually in frameworks like Sora. A benefit of our compiler is that it automatically vectorizes programs to operate on arrays, similarly to [9, 26]. The goal of this transformation is to rewrite a computation of type $\mathsf{ST}\ t\ a\ b$ to one of type $\mathsf{ST}\ t\ (\mathsf{array}[N]\ a)\ (\mathsf{array}[M]\ b)$. This process consists of three steps:

1. First, an analysis identifies the number of values that a computer (one of type $\mathsf{ST}\ (\mathsf{C}\ \nu)\ a\ b$) takes from the input stream (call this $\alpha_{\text{in}}$) and emits to the output stream (call this $\alpha_{\text{out}}$) before returning. We call this the *cardinality* information.
2. Next, for every repeat $c$ with an identified cardinality for $c$ we take the union of two candidate sets of possible vectorizations: The first set comes from *scale-up* vectorization: All candidates in this set have types of the form $\mathsf{ST}\ (\mathsf{C}\ \nu)\ (\mathsf{array}[k * m * \alpha_{\text{in}}]\ a)\ (\mathsf{array}[m * \alpha_{\text{out}}]\ b)$. We allow components to take input arrays that are any multiple $(k * m)$ of the input cardinality, but restrict the output array so that the multiplicity $(m)$ is divisible by the output multiplicity. We do not vectorize the output to *any* multiplicity of the output cardinality to avoid situations where the output array is only half-filled but there is no more data to process on the input. The second set comes from *scale-down* vectorization, which only applies if the component has a large $\alpha_{\text{in}}$ or $\alpha_{\text{out}}$ cardinality (not uncommon in the Wifi pipelines). Scale down vectorization will create a transformed computation

[3] Defined using the times combinator, a variation of repeat that repeats a computation a finite number of times.

that has type ST (C $\nu$) (array[$d_{\text{in}}$] $a$) (array[$d_{\text{out}}$] $b$), for some divisor $d_{\text{in}}$ and $d_{\text{out}}$ of $\alpha_{\text{in}}$ and $\alpha_{\text{out}}$ respectively.

3. Once we have identified scale-up and scale-down sets for computations in the pipeline, we must compose them across the bind and arrow operators in the program in a way that maximizes performance. Note that our optimization should aim to increase vectorization sizes across all components equally, as the smallest batch size is likely to be a bottleneck. In order to achieve this balance, we use the optimization framework from [20], which is well-studied in the context of networking. Each arrow and bind operator is assigned a *utility* number, which is obtained by applying a predefined concave function[4] to the corresponding vectorization size. We attempt to find a composition of vectorizations that respects type correctness and maximizes the sum of all utilities. One key benefit of the approach from [20] is that by maximizing the sum of concave utilities we balance the optimization across all batch sizes. The other key benefit is that this can be done very efficently, in a greedy manner.

The following is the automatically vectorized version of the scrambler from Listing 1 where input and output types have been converted to arrays of 8-bits that can be conveniently packed into chars in the generated C code.

```
1  let vectorized_scrambler (u: unit) =
2   letref scrmbl_st: arr[7] bit := {1,1,1,1,1,1,1};
3          tmp:bit; y:bit
4   in repeat
5    (let up_wrap_17 () =
6      letref ya_19: arr[8] bit in
7        (xa_18 : arr[8] bit) ← take;
8        (times 8 (\j_21.
9          x ← return xa_18[0*8+j_21*1+0];
10         return (
11           tmp := scrmbl_st[3]^scrmbl_st[0];
12           scrmbl_st[0:+6] := scrmbl_st[1:+6];
13           scrmbl_st[6] := tmp;
14           y := x^tmp);
15         return ya_19[j_21*1+0] := y));
16       emit ya_19
17    in up_wrap_17())
```

**Listing 4.** Auto-vectorized scrambler

Of course this is a relatively complicated computation that includes many sub-computations, but fortunately the optimizer shines here: Post-vectorization and post-optimization, this program is *automatically* converted to use a tight expression-level for-loop and the whole computation is auto-mapped into a single **map** computation.

### 4.3 Lookup table generation

Lookup tables (LUTs) are used pervasively in Sora for performance. However, as Section 2 demonstrated when comparing Sora's lookup table-based implementation of the scrambler (Listing 2) to the Ziria version (Listing 1), writing functions that use LUTs leads to code that is difficult to write, read, and modify. Furthermore, the size of the LUT may depend on the outcome of other optimizations such as vectorization, leading to frequent LUT recomputation.

Ziria frees the programmer from having to forsake readability for performance—and from the pain of generating LUTs by hand—by providing compiler support for transparently compiling high-level functions to LUTs. Functions may be hand-annotated with the keyword lut, or the programmer may direct the compiler to *automatically* detect portions of a Ziria program that are amenable to a LUT implementation. In this case, the compiler looks for expressions that are complex enough to be worth converting to a LUTs, but whose LUTs sizes are reasonable.

[4] In our implementation we use $\log(\cdot)$, as in [20].

For instance, our auto-LUT analysis automatically identifies that the body of the auto-mapped function obtained by further optimization of Listing 4 has inputs of total bitwidth 15 (scrambl_st, and xs_18) and outputs of total bitwidth 25 (8 for the result of the function, 8 for ya_19, 7 for scrmbl_st, and 2 for tmp and y), and automatically creates the corresponding ~100K LUT. Moreover, Ziria includes a bit permutation primitive, bperm, which is automatically converted to a LUT if the permutation table is statically known. bperm is useful in wireless interleavers, which reduce errors in signal transmissions by applying pseudorandom permutations.

### 4.4 Pipelining

As explained in Section 3, when two Ziria computations $c_1$ and $c_2$ are connected by $\ggg$, they cannot mutate shared state and hence can be pipelined onto multiple cores on SMP platforms. To support this kind of parallelization the Ziria compiler allows user annotations of type $c_1 |\ggg| c_2$. This code is automatically pipelined onto two cores by (i) allocating a fresh single-reader single-writer queue $q$, (ii) transforming the code to $c_1 \ggg \mathsf{write}(q) \,|\ggg|\, \mathsf{read}(q) \ggg c_2$, then (iii) spawning two threads: $c_1 \ggg \mathsf{write}(q)$, which is pinned to physical core 1, and $\mathsf{read}(q) \ggg c_2$, which is pinned to physical core 2. The design of synchronization queues is adapted from Sora [29].

## 5. Evaluation

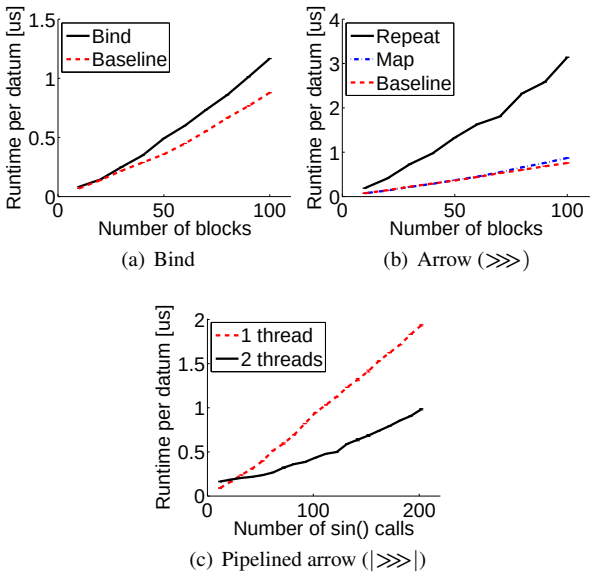
In this section we seek to answer the following questions about the performance of Ziria: (1) What is the overhead of Ziria's execution engine (2) What is the speedup of various compiler optimizations? (3) How does Ziria compare with state-of-the-art implementations of a real-world wireless protocol? To answer these questions we perform two sets of measurements. First, we use a number of microbenchmarks to quantify the overheads of Ziria combinators bind, arrow ($\ggg$), and pipelined arrow ($|\ggg|$). Then we present an implementation and a detailed evaluation of WiFi (802.11a/g) in Ziria. We show the speedups achievable with the Ziria compiler's automatic vectorizer, lookup table generator, and annotation-guided pipeliner, and associated overheads on a real-world wireless physical layer protocol. As a baseline for comparison, we use Sora [29], one of the few CPU-based SDR platforms with a line-rate implementation of a full WiFi PHY.

### 5.1 Methodology

We evaluate the Ziria framework on a Dell T3600 PC with an Intel Xeon E5-1620 CPU at 3.6 GHz, running Windows. In order to compare our performance results to Sora, we use the same C compiler as Sora (Windows Driver Development Kit version 7) and we adapt our runtime to use Sora's runtime libraries. In particular, we use Sora's user-mode threading library, which allows us to run our user-mode threads at the highest priority and pinned to a specific core, effectively preventing the OS from preempting the execution.

We evaluate our framework on various DSP algorithms that are part of a standard WiFi transceiver. Some of these algorithms operate on bits (*e.g.* most of the TX) and some on complex samples (*e.g.* most of the RX). The throughput of each algorithm is proportional to the sample width in bits. However, the overhead of Ziria execution model mainly depends on the number of data items being processed and not their actual widths. Therefore, throughout this section we present the throughput in units of Md/s (mega data per second).

Sora hardware, as well as the hardware of other high-performance SDR platforms, is designed to mitigate I/O overheads. The Sora software component fetches radio samples through PCI bus at a speed comparable to main memory reads. In our evaluation we focus on measuring the performance of the software component. When we evaluate the components' performance we read input samples directly from the memory and discard them at the output.




(a) Bind (b) Arrow (⋙) (c) Pipelined arrow (|⋙|)

**Figure 9.** Overheads of various Ziria components.

### 5.2 Microbenchmarks

***Bind*** We first measure the overhead of bind. We do this by measuring the runtime of a program containing $n$ computation components bound together in sequence, with each component consisting of a single call to `sin()`. As a performance baseline, we use the runtime of a semantically equivalent program in which all $n$ `sin` operations are executed in a single component.

The results are depicted in Figure 9(a). The dashed line gives runtime of the baseline program for $n$ `sin` computations. The solid line gives runtimes for $n$ `sin` components bound in sequence. We ran each program 10 times on 20 million inputs and report average execution time per data item (confidence intervals are very small). We verify that the runtime data fit a linear model as a function of $n$, indicating that the cost of bind grows linearly with the number of components. The cost of a single bind operation on our system, given by the difference of the slopes of the two lines, is around 3ns.

***Arrow*** Next, we measure the cost of arrow (⋙). To do so we measure the runtime of a program containing $n$ components composed with ⋙, with each component repeating the computation (`x←take; emit(sin(x))`). This experiment is labeled *Repeat*. We also measured the runtime of an optimized version of this program that uses `map` computation `map(sin(x))` in the experiment labeled *Map*. As a baseline, we use the runtime of a program in which all $n$ `sin` calls are merged into a single component. Again, we ran the code 10 times on 20 million inputs each time and report average execution time per data item (confidence intervals are very small).

The results are depicted in Figure 9(b). The cost of a single repeat arrow operation on our system, given by the difference of the slopes of the Repeat and Baseline lines, is around 24ns, and the cost of a single map arrow is around 1ns. This difference stems from the fact that the **`repeat`** computation has to execute several ticks and procs in each round, whereas the **`map`** execution is completely streamlined. However, as discussed in Section 4.1, our high-level optimizations are often able to transform a **`repeat`** computation into a **`map`** computation, and mitigate this overhead. The overhead is typically further amortized over multiple inputs due to vectorization.

***Pipelined arrow*** To gauge the overhead of pipelining Ziria programs onto multiple cores using |⋙|, we measured the runtime of $n$ `sin` calls when (i) run on a single core, and (ii) divided evenly onto two cores. Plot 9(c) shows the results of this experiment. The red dashed line and the solid black line give the execution time per datum in microseconds when running on a single core and on two cores, respectively. The point at which these two lines intersect, approximately 30 computations per datum, is the point at which we break even, *i.e.*, when pipelining gives speedup rather than slowdown. Furthermore, speedup is approximately 1.7x at 60 calls and 2x at 90 calls. As we show later in the WiFi evaluation, most signal processing blocks are computationally intensive, and the pipelining benefits are on the high side.

### 5.3 Performance of WiFi 802.11a/g

This section evaluates the performance of our Ziria implementation of the physical layer of 802.11a/g, the most popular variant of the WiFi protocol. Our implementation consists of ≈ 3k lines of Ziria code in total. We evaluate data throughput of a number of the processing blocks in our WiFi transmitter (TX) and receiver (RX), as well as end-to-end performance of both TX and RX.

We compare our Ziria implementation with the manually optimized Sora implementation [29] and the WiFi requirements. We have first verified the correctness of each Ziria block against the corresponding block in the Sora implementation. We then profile each block in both implementations by sending them the same input data.

Different implementations may use different signal processing algorithms in an effort to improve the receiver's performance. To allow for a fair comparison, our WiFi implementation uses the same receiver algorithms as the Sora implementation.

As a part of Ziria we provide a basic signal processing library. This library provides a high-level interface for very efficient implementations (borrowed from Sora) of three common signal processing blocks: FFT, IFFT and Viterbi. These blocks are standardized and reused across all modern physical layers (WiFi, WiMax, LTE), and their efficient implementations are already available across a large range of SDR platforms. The library also includes a basic SIMD library for vector operations that operates on Ziria's basic types.

In the case of the end-to-end transmitter and receiver code, different input data may activate different control paths in the code. We profile these various code paths separately, both for Sora and Ziria implementations. In particular, we profile the signal detection path (CCA) at the receiver that is running when the receiver is scanning for a packet, as well as the transmitter and the receiver at different data rates (6, 12, 24, and 48 Mbps).

#### 5.3.1 Receiver

We start by profiling the receiver's building blocks. In order to asses the performance of our vectorization algorithm, we compile each block with no vectorization, manual vectorization copied from the Sora implementation, and the automatic vectorization described in Section 4.2. We also enable lookup table generation, but it does not play a major role in the receiver since most of the receiver blocks operate on complex samples and cannot be converted into lookup tables. The results are given in Table 1 and Table 2.

Although Sora's WiFi implementation is manually tuned for high performance, all of the blocks in our high-level Ziria implementation are less than two times slower than the corresponding Sora blocks, and many of them approach the performance of their Sora counterparts. Furthermore, all our end-to-end code paths satisfy the WiFi specification requirements of 40 Msamples per second.

We also observe that vectorization can significantly improve the performance of Ziria code, often by more than an order of magnitude. We see that the performance of our automatic vectorization comes close to the performance of the manually vectorized Ziria code in which we use Sora's vectorization annotation.

| Block | Sora [Md/s] | Auto vect. [Md/s] | Non vect. [Md/s] | Manual vect. [Md/s] |
|---|---|---|---|---|
| DownSample2 | *1139* | **668** | 81 | 612 |
| RemoveDC | *644* | **1544** | 57 | 925 |
| CCA | *193* | **130** | 66 | 131 |
| LTS | *364* | **183** | 74 | 183 |
| DataSymbol | *4775* | **2588** | 72 | 4201 |
| FreqCompensation | *1588* | **1304** | 63 | 1465 |
| FFT | *465* | **392** | 58 | 411 |
| ChannelEqualization | *1680* | **1007** | 64 | 1404 |
| PhaseCompensate | *1716* | **1302** | 63 | 1496 |
| PilotTrack | *435* | **246** | 52 | 360 |
| Viterbi 1/2 | *165* | **144** | 84 | 149 |
| Descrambler | *289* | **370** | 108 | 425 |
| DemapBPSK | *866* | **926** | 82 | 932 |
| DeinterleaveBPSK | *2943* | **1496** | 104 | 1920 |
| DemapQPSK | *761* | **714** | 49 | 648 |
| DeinterleaveQPSK | *3121* | **1391** | 105 | 1444 |
| DemapQAM16 | *633* | **503** | 35 | 567 |
| DeinterleaveQAM16 | *3312* | **1496** | 106 | 1742 |
| DemapQAM64 | *522* | **194** | 30 | 226 |
| DeinterleaveQAM64 | *3365* | **1485** | 106 | 1597 |

**Table 1.** Throughputs of different blocks in WiFi receiver

| Block | Sora [Md/s] | Auto vect. [Md/s] | WiFi [Md/s] | Non vect. [Md/s] | Manual vect. [Md/s] |
|---|---|---|---|---|---|
| RX 6Mbps | *164* | **91** | 40 | 20 | 115 |
| RX 12Mbps | *125* | **66** | 40 | 16 | 78 |
| RX 24Mbps | *81* | **50** | 40 | 14 | 57 |
| RX 48Mbps | *61* | **40** | 40 | 12 | 45 |
| RX CCA | *289* | **163** | 40 | 55 | 181 |

**Table 2.** Throughputs of end-to-end code paths of Wifi receivers

#### 5.3.2 Transmitter

We next profile the performance of the WiFi transmitter. In the transmitter case, lookup table generation can significantly affect the performance of the code. Therefore, we compare a Ziria implementation of the transmitter without any optimization (no lookup tables, no vectorization), an auto-vectorized Ziria implementation of the transmitter without lookup tables, and a fully optimized Ziria WiFi transmitter. The results are given in Tables 3 and 4.

Again, we see that our optimizations significantly speed up the code, often by more than an order of magnitude. For some blocks our code is close to the performance of Sora. Other blocks produce significant slow-down. This slow-down happens only for very fast blocks with more than 1Gd/s (on a CPU running at 3.6 GHz), where even a few extra memory operations introduced by the Ziria compiler greatly affect performance. But these blocks are so fast that they are not bottlenecks themselves. As can be seen in 4, the end-to-end transmitter is 2x-4x slower than Sora, but still comfortably above the WiFi requirements for all code paths.

#### 5.3.3 Pipelining

One of the great potentials of Ziria is seamless pipeline parallelization of the code. In this section we evaluate the performance of the pipelined arrow operator on the WiFi code base. The results are given in Table 5. We manually find the optimal split for each example. The performance of the pipelined case is limited by its slowest part, which is the Viterbi decoder in the receiver and the IFFT in the transmitter. The two parts are more balanced at low data rates, hence we observe almost 90% speed-up at 6 Mbps.

| Block | Sora [Md/s] | LUT, vect. [Md/s] | No LUT, no vect. [Md/s] | No LUT, vect. [Md/s] |
|---|---|---|---|---|
| IFFTx | *207* | **157** | 31 | 157 |
| AddPilot | *759* | **1237** | 51 | 1224 |
| Map11aBPSK | *2870* | **1534** | 147 | 168 |
| Map11aQPSK | *3788* | **1551** | 129 | 207 |
| Map11aQAM16 | *2923* | **1895** | 126 | 237 |
| Map11aQAM64 | *2796* | **2770** | 134 | 352 |
| InterleaveBPSK | *4888* | **238** | 89 | 170 |
| InterleaveQPSK | *3805* | **346** | 74 | 126 |
| InterleaveQAM16 | *2026* | **530** | 80 | 224 |
| InterleaveQAM64 | *1423* | **639** | 82 | 298 |
| ConvEncode 12 | *3062* | **342** | 33 | 42 |
| ConvEncode 23 | *2589* | **590** | 38 | 46 |
| ConvEncode 34 | *3722* | **306** | 35 | 45 |
| Scrambler | *3781* | **843** | 57 | 59 |

**Table 3.** Throughputs of different blocks in WiFi transmitter

| Block | Sora [Md/s] | LUT, vect. [Md/s] | WiFi [Md/s] | No LUT, no vect. [Md/s] | No LUT, vect. [Md/s] |
|---|---|---|---|---|---|
| TX 6 | *54* | **27** | 6 | 5 | 12 |
| TX 12 | *98* | **35** | 12 | 7 | 13 |
| TX 24 | *145* | **51** | 24 | 9 | 15 |
| TX 48 | *231* | **66** | 48 | 11 | 17 |

**Table 4.** Throughputs of different code paths of Wifi transmitter

| RX | 1 th. | 2 th. | WiFi | TX | 1 th. | 2 th. | WiFi |
|---|---|---|---|---|---|---|---|
| 6Mbps | *88* | **156** | 40 | 6Mbps | *27* | **51** | 6 |
| 12Mbps | *65* | **100** | 40 | 12Mbps | *35* | **45** | 12 |
| 24Mbps | *48* | **67** | 40 | 24Mbps | *51* | **53** | 24 |
| 48Mbps | *40* | **52** | 40 | 49Mbps | *66* | **70** | 48 |

**Table 5.** Throughputs of WiFi transmitter and receiver with and without pipelining. All throughputs are expressed in Md/s.

## 6. Related work

***Software-defined radio*** Existing SDR platforms can be divided roughly into two groups: FPGA-based [23, 24] and processor-based [1, 2, 5, 29]. Processor-based platforms are popular because of their low cost and programmability. They typically use programming tools and abstractions that are appealing to wireless engineers (*e.g.* C and Python), but require extensive and difficult optimizations to achieve the line speeds of modern PHYs. Otherwise, like GNURadio [10], they have limited use for testing and experimentation.

Recent programming frameworks for Sora (Bricks [3]) and GnuRadio (blocks [10] and VOLK [27]), as well as platform-independent frameworks such as OpenRadio [6] and CODIPHY [15], help programmers extract the most from existing hardware, typically by providing libraries of hand-optimized DSP blocks. While these frameworks provide useful resources to programmers, the fact that they are implemented as libraries means that they impose more manual configuration and optimization on the user than is typical in Ziria. Bricks [3], for example, is a library of hand-optimized blocks that can be connected with C++ templates to form processing pipelines. Building Brick processing pipelines often requires manually vectorizing a number of blocks in the processing chain, in order to match the vectorization widths of connected components. Sora Bricks often include manually generated lookup tables for

performance. Similar considerations apply to VOLK, in which the programmer must choose and manually configure the vectorization widths of each component in a GNURadio pipeline. In Ziria, we avoid this sort of manual configuration by automatically vectorizing and generating lookup tables for compiled Ziria code. The result is high-level code close to the standard specification that still runs fast.

***Dataflow languages*** In addition to work on SDR, Ziria builds on a significant body of programming languages research. Synchronous dataflow languages [8, 11, 12] have been used in embedded and reactive systems for modeling and verification but—to our knowledge—never to implement line-rate software PHY designs. StreamIt [31], also based on synchronous dataflow, was one of the early works to target DSP applications in software. Example programs in StreamIt include software radios, like Wifi and 3GPP PHY. StreamIt demonstrated that a DSL can enable significant optimizations, and emphasized on multicore execution [17].

Ziria learns from StreamIt and makes improvements in several respects. First, StreamIt programs are graphs of independent filters. To express *control* dependencies between nodes in the graph one uses splitter (demultiplexing) blocks, or *teleport messages* [32], a form of asynchronous message passing with guaranteed logical delivery bounds. Ziria's explicit treatment of control fortunately helps here. An example of the need for teleport messaging in StreamIt comes from frequency hopping (Figure 12 [32]), which can be expressed in Ziria as:

```
1  aToD >>> letref freq : float = startFreq in repeat (
2    newFreq ← rfToIf()>>>fft()>>>checkFreqHop(freq);
3    return (freq := newFreq))
```

The checkFreqHop is a *computer* that eventually returns a new frequency after emitting some elements with the old frequency. Ziria's treatment of shared state with the restricted typing of >>> exposes pipeline parallelization opportunities. Ziria also exposes state (re-)initialization and associates it with bind reconfiguration.These features are in accordance with lessons distilled from the StreamIt experience [30]. Unlike StreamIt, Ziria does not support data parallelism, which appears to be less applicable in wireless PHY design due to heavy control dependencies in processing pipelines. Finally, although there is a StreamIt WiFi implementation, it is unknown whether it can be deployed at line rates.

***Functional programming and functional reactive programming*** Ziria builds on a broad range of techniques from functional programming. The design of Ziria and its key combinators and their optimization draw from monads and arrows [18, 22]. A flavor of our bind combinator (called "switch") can be found in Yampa [13], a popular functional reactive programming (FRP) framework. However, Yampa encodes control information onto the data channel, whereas Ziria keeps these notions separate. The Ziria's tick and process semantics bears some resemblance to the push and pull model of computations. Typically FRP uses the pull model, but there exists recent work on combining the two to improve efficiency [16].

The vectorization transformation is inspired by work in nested data parallelism [9, 26]. Ziria's stream computation semantics was influenced by the work on stream fusion. For example, the result type we use to implement a single step of stream computation in the runtime model of Section 3 is effectively the same as that used to represent streams in Coutts et al. [14].

Finally, there exists recent work on using high-level functional languages for DSP applications that aspires to match the efficiency of low-level implementations [4]. In principle, such a language could replace the imperative sub-language inside Ziria blocks.

# 7. Conclusion

We presented Ziria, the first high-level SDR platform with a performant execution model. To validate our design, we built a compiler that performs optimizations done manually in existing CPU-based SDR platforms: vectorization, lookup table generation, and annotation-guided pipelining. To demonstrate Ziria's viability, we used Ziria to build a rate-compliant PHY-layer for WiFi 802.11a/g.

# References


[1] Texas Instruments TMS320TCI6616 Communications Infrastructure KeyStone SoC.

[2] Ettus Research, USRP: Universal Software Radio Peripheral.

[3] Microsoft Research, Brick specification, 2011.

[4] E. Axelsson et al. Feldspar: A domain specific language for digital signal processing algorithms. In *FMMC*, 2010.

[5] H. V. Balan et al. USC SDR, an easy-to-program, high data rate, real time software radio platform. In *SRIF*, 2013.

[6] M. Bansal et al. OpenRadio: a programmable wireless dataplane. In *HotSDN*, 2012.

[7] T. Bansal et al. Symphony: Cooperative packet recovery over the wired backbone in enterprise WLANs. In *MOBICOM*, 2013.

[8] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *SCP*, 19(2), 1992.

[9] G. E. Blelloch and G. W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *JPDC*, 8(2):119–134, Feb. 1990.

[10] E. Blossom. GNURadio: tools for exploring the radio frequency spectrum. *Linux Journal*, 2004(122):4, 2004.

[11] P. Caspi. Lucid Synchrone. In *Actes du colloque INRIA OPOPAC, Lacanau*. HERMES, November 1993.

[12] P. Caspi and M. Pouzet. Lucid Synchrone, a functional extension of Lustre. Technical report, Univ. Pierre et Marie Curie, Lab. LIP6, 2000.

[13] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Haskell Workshop*, 2001.

[14] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *ICFP'07*, 2007.

[15] A. Dutta, D. Saha, D. Grunwald, and D. Sicker. CODIPHY: Composing on-demand intelligent physical layers. In *SRIF*, 2013.

[16] C. Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

[17] M. I. Gordon. *Compiler Techniques for Scalable Performance of Stream Programs on Multicore Architectures*. PhD thesis, 2010.

[18] J. Hughes. Generalising monads to arrows. *SCP*, 37(1-3), 2000.

[19] IEEE. Part 11: Wireless LAN MAC and PHY specifications high-speed physical layer in the 5 GHz band, 1999. URL http://standards.ieee.org/getieee802/download/802.11a-1999.pdf.

[20] F. P. Kelly. Charging and rate control for elastic traffic. *European Transactions on Telecommunications*, 8:33–37, 1997.

[21] T. Li et al. CRMA: Collision-resistant multiple access. In *MOBICOM*, 2011.

[22] E. Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1), 1991.

[23] P. Murphy, A. Sabharwal, and B. Aazhang. Design of WARP: a wireless open-access research platform. In *ESPC*, 2006.

[24] M. C. Ng et al. Airblue: a system for cross-layer wireless protocol development. In *ANCS*, 2010.

[25] G. Nychis et al. Enabling MAC protocol implementations on software-defined radios. In *NSDI*, 2009.

[26] S. Peyton Jones. Harnessing the multicores: Nested data parallelism in Haskell. In *APLAS*, 2008.

[27] T. Rondeau et al. SIMD programming in GNURadio: Maintainable and user-friendly algorithm optimization with VOLK. In *SDR WinnComm*, 2013.

[28] S. Sen, R. R. Choudhury, and S. Nelakuditi. No time to countdown: Migrating backoff to the frequency domain. In *MOBICOM*, 2011.

[29] K. Tan et al. Sora: high performance software radio using general purpose multi-core processors. In *NSDI*, 2009.

[30] W. Thies and S. Amarasinghe. An empirical characterization of stream programs and its implications for language and compiler design. In *PACT'10*, 2010.

[31] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: a language for streaming applications. In *Compiler Construction*, 2002.

[32] W. Thies et al. Teleport messaging for distributed stream programs. In *PPoPP'05*, 2005.